Introducing Parallelism to the Ranges TS

Gordon Brown
Codeplay Software Ltd
gordon@codeplay.com

Christopher Di Bella
Codeplay Software Ltd
christopher@codeplay.com

Michael Haidl
University of Münster
michael.haidl@uni-muenster.de

Toomas Remmelg
Codeplay Software Ltd
toomas.remmelg@codeplay.com

Ruyman Reyes
Codeplay Software Ltd
ruyman@codeplay.com

Michel Steuwer
University of Glasgow
michel.steuwer@glasgow.ac.uk

Abstract

The current interface provided by the C++17 parallel algorithms poses some limitations with respect to parallel data access and heterogeneous systems, such as personal computers and server nodes with GPUs, smartphones, and embedded System on a Chip chipsets. In this paper, we present a summary of why we believe the Ranges TS solves these problems, and also improves both programmability and performance on heterogeneous platforms.

The complete paper has been submitted to WG21 for consideration, and here we present a summary of the changes proposed alongside new performance results.

To the best of our knowledge, this is the first paper presented to WG21 that unifies the Ranges TS with the parallel algorithms introduced in C++17. Although there are various points of intersection, we will focus on the composability of functions, and the benefit that this brings to accelerator devices via kernel fusion.

Keywords C++, Parallel Programming, Heterogeneous Computing, Kernel Fusion

ACM Reference Format:
Gordon Brown, Christopher Di Bella, Michael Haidl, Toomas Remmelg, Ruyman Reyes, and Michel Steuwer. 2018. Introducing Parallelism to the Ranges TS. In *IWOCL '18: International Workshop on OpenCL (IWOCL '18), May 14–16, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 5 pages.

IWOCL '18, May 14–16, 2018, Oxford, United Kingdom

1 Contrasting Ranges, Views, and Actions with Iterators

As already presented in P0687R0 [12], the limitations of the iterator-based interface for Standard algorithms have a significant performance impact on systems where memory must be copied before applying the algorithms to the data, which can be problematic. A potential solution to this problem is to use the interface introduced in the Ranges TS [8] and Eric Niebler's current proposal for range adaptors and utilities. [10] Among several other modifications, the range-based algorithms' interface replaces iterator-pairs with ranges. A range is a concept that defines the requirements of a type that allows iteration over its elements by providing a begin iterator and a sentinel object. [11] In a general context, a range is typically a container, but is not required to be one (for example, an input iterator abstracting over `std::istream` is a range if it is associated with some object denoting the end of the input sequence).

The Ranges TS also revises the C++14 Standard algorithms, so that they provide iterator-sentinel pairs instead of homogeneously-typed iterator-pairs. This means that we no longer need to specify the end of a range using an iterator, and can use other objects instead; provided that the object has some sort of tangible relationship with our iterator type. One such example where this is useful is doing something to the first `n` elements of a range. The C++ standard library approximates this with functions such as `std::fill_n`, but completeness requires providing this for every algorithm, and that is an enormous number of additional overloads that need to be ratified, implemented, confirmed to be correctly implemented, and maintained. [2] An example of adding such an overload to `std::find` using standard algorithms can be seen in Listing 1. The Ranges TS version in Listing 2 more clearly communicates the fact that you only want to move forward `count` number of steps.

Eric Niebler's *range-v3* [9] library is a cross-platform, experimental implementation of the Ranges TS, and is compatible with C++11. Unlike the Ranges TS reference implementation, *cmcstl2* [4], *range-v3* does not rely on the Concepts TS, and is thus suitable for use with implementations that don't support Concepts.

```
1  template <typename ForwardIterator, typename N,
2            typename T>
3  ForwardIterator find_n(ForwardIterator first,
4      N count, T const& value)
5  {
6    return std::find(first,
7        std::next(first, count), value);
8  }
```

Listing 1. `find_n` implementation using Standard algorithms.

```
1  namespace ranges = std::experimental::ranges;
2  template <ranges::InputIterator I, typename T>
3  I find_n(I first, ranges::difference_type_t<I>
4           count, T const& value)
5  {
6   return ranges::find(
7     ranges::make_counted_iterator(first, count),
8     ranges::default_sentinel{}, value);
9  }
```

Listing 2. `find_n` implementation using the Ranges TS.

,

```
1  std::vector<float> x = // ...
2  std::vector<float> y = // ...
3  float a =              // ...
4
5  auto out = std::vector<float>(size(x));
6  {
7    auto temp = std::vector<float>(size(x));
8    std::transform(begin(x), end(x), begin(temp),
9        [a](const auto x) { return a * x; });
10
11   std::transform(begin(temp), end(temp),
12       begin(y), begin(out), std::plus<>{});
13 }
```

Listing 3. Slow-path `saxpy` implementation using STL.

Central to *range-v3* are views and actions, which improve algorithm composability, and allow users to construct pipelines of operations using `operator|`. Views behave as range-based algorithms; but unlike algorithms, lazily perform non-modifying computation only when requested. Actions represent mutating operations and perform in-place modifications to ranges. P0789 [10] proposes adding views to the Ranges TS.

2 An Example Use-Case

Given two vectors, $\mathbf{x}$ and $\mathbf{y}$, and a scalar α, the result of the BLAS (Basic Linear Algebra Subprograms) primitive, `saxpy`, is defined as $\alpha\mathbf{x}+\mathbf{y}$. A natural way of writing this using the STL is to scale $\mathbf{x}$ by α using `std::transform`, and then add the scaled vector with $\mathbf{y}$ using a second call to `std::transform` as in Listing 3.

```
1  std::vector<float> x = // ...
2  std::vector<float> y = // ...
3  float a =              // ...
4
5  auto out = std::vector<float>(size(x));
6  {
7    std::transform(begin(x), end(x), begin(y),
8        begin(out), [a](auto x, auto y) {
9          return a * x + y; });
10 }
```

Listing 4. Fast-path `saxpy` implementation using a single transformation with addition.

Notice the use of the temporary vector `temp` in line 7; this can be expensive on accelerators. The temporary vector not only requires additional memory, but the executing code also performs additional stores and loads to access the temporary which can hinder performance. To avoid the temporary variable, the scaling operation can be manually done in a single transformation operation together with the addition as seen in Listing 4.

A similar problem arises when performing the dot-product operation on two vectors. To avoid a temporary variable, the programmer needs to be aware that `std::inner_product` is a more suitable alternative to a combination of `std::transform` and `std::accumulate`.

However, there is not always a predefined function – such as `std::inner_product` – available for all the possible cases that application developers can concoct with Standard algorithms. For example, `std::transform` and `std::transform_reduce` are limited to one or two input ranges. If a user wants to combine more ranges in a single call, they are required to convert the input data to be formatted as an array of structures (AoS), and then apply the algorithm.

On CPUs, the AoS format hinders the compiler's ability to perform vectorisation – as allowed by the `std::execution::par_unseq` execution policy – since vector registers can typically only hold homogeneous data. Thus, the conversion might have negative performance implications. On GPUs, this format causes uncoalesced memory-accesses, since threads will be performing non-contiguous accesses as shown in Figure 1.

Range-based algorithms and views enable developers to compose the provided algorithms more flexibly, and simultaneously offer the opportunity to increase performance by eliminating temporary storage. The lazy nature of views also enables automatic fusion of multiple algorithms into a single, efficient, computational kernel when using heterogeneous systems. By providing a compile-time function-composition mechanism, device-kernels can be generated to minimise register pressure on custom algorithms that have been written by developers. This kernel-fusion technique is widely used in libraries,

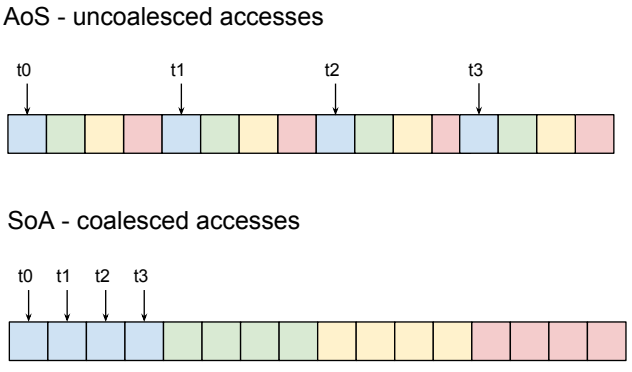

Figure 1. Access pattern differences on GPU memory when using AoS vs SoA. Coalesced access is preferred for performance reasons.

```
1 std::vector<float> x = // ...
2 std::vector<float> y = // ...
3 float a =              // ...
4
5 auto ax = ranges::view::zip(view::repeat(a), x)
6         | ranges::view::transform(mult);
7 auto out = ranges::view::zip(ax, y)
8          | ranges::transform(plus)
9          | ranges::to_vector;
```

Listing 5. `saxpy` implementation using *range-v3*.

such as Eigen [6], to improve performance of various computational kernels.

A range-based interface also mitigates the problem of ensuring that an iterator-pair addresses the same range. A range has a beginning and a sentinel object to describe its termination, and its contents can be transparently converted using range-based actions and views.

`saxpy` can now be implemented as shown in Listing 5. The variable `out` now contains a view of the computation result and no computation has yet been performed. The result can be materialised by copying it to a `std::vector` when necessary. Accessing `out` will perform the actual computation.

3 Implementation Prototype of Parallel STL with Ranges

We have implemented a prototype [1, 3] of some parallel algorithms using *range-v3* and SYCL. This builds upon work using views and actions with PACXX. [7]

`saxpy` is then implemented as it is in Listing 6. We first create a SYCL execution policy, `exec`, in line 7. It provides parallel implementations of Standard algorithms, and wraps a SYCL queue attached to a device, so that it can enqueue work. `std::experimental::copy` internally creates a SYCL buffer that keeps track of the

```
1  std::vector<float> x = // ...
2  std::vector<float> y = // ...
3  float a =              // ...
4
5  std::vector<float> out(size(x));
6  {
7    gstorm::sycl_exec exec;
8
9    using std::experimental::copy;
10   auto ax = ranges::view::zip(
11       ranges::view::repeat(a), copy(exec, x))
12           | ranges::view::transform(mult);
13   std::experimental::transform(exec,
14       ranges::view::zip(ax, copy(exec, y)),
15       copy(exec, out), plus);
16 }
```

Listing 6. `saxpy` implemented using SYCL and *range-v3*.

usage of memory, and holds all the required metadata. The SYCL buffer then provides access to the data from the different accelerators via accessors. Our prototype implements a wrapper that provides an `InputRange`-compatible interface for SYCL buffers to allow them to be used as input to views.

Since views are lazy and don't perform any computation, we change the final `ranges::view::transform` to a call to `std::experimental::transform` on the execution policy, to execute the resulting operation and enqueue the kernel onto the device. It is important to understand that this code will only execute a single kernel on the device, despite the use of four view algorithms to describe the computation. As views never execute eagerly, and only algorithms and action do, this gives a very easy to understand cost model to the programmer.

The lifetime of SYCL objects ends at the end of the enclosing scope; following their lifetime rules, any data modified on the device will be updated on the host.

Views from *range-v3* access data using iterators from the provided input range. To access device memory, iterators need to use SYCL accessors when dereferenced. We implement a SYCL-aware wrapper, `gvector`, that allocates a SYCL buffer, and registers itself with the execution policy. The execution policy will provide registered `gvector`s with `cl::sycl::handler`s when launching a kernel, so that they can create accessors to be used in device code, and return iterators from them using `begin` and `end`. The authors intend to propose to extend the SYCL buffer to implement the behaviour currently implemented by `gvector`.

The limitations of SYCL, imposed by the nature of heterogeneous dispatch and multiple device support, required making changes to *range-v3*:

- Non-standard-layout `std::tuple` was replaced with an implementation that does.

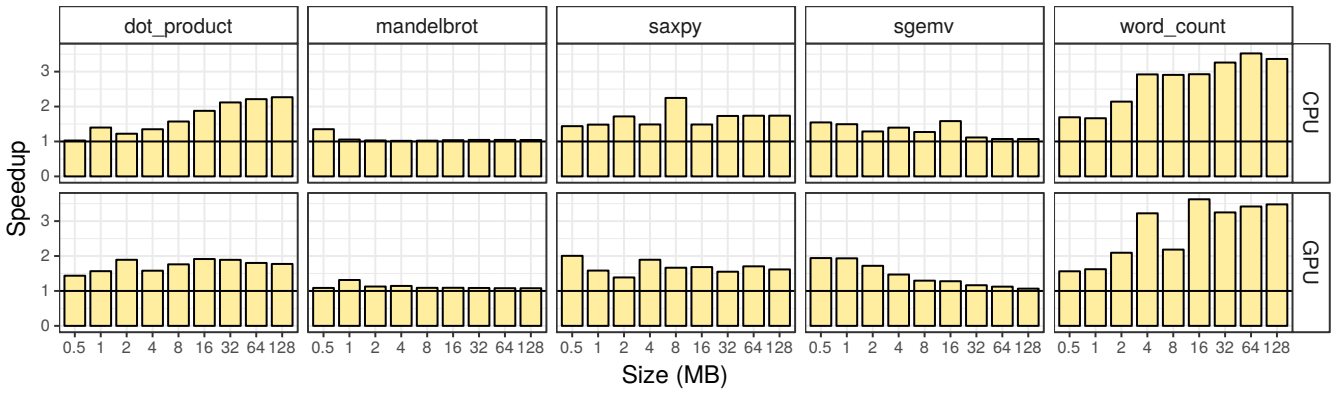

Figure 2. Speedups from automatc kernel fusion using views.

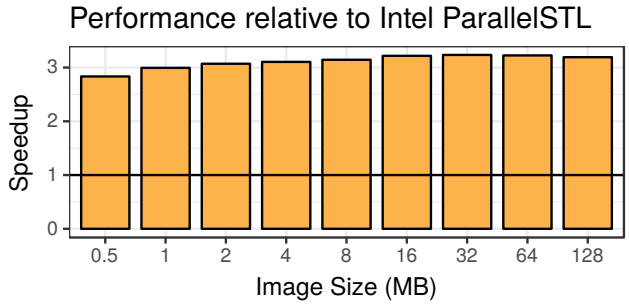

Figure 3. Speedup on an Intel i7-6700K CPU.

- Making `ranges::view::chunk` standard-layout by removing `ranges::box` base class.
- Making `ranges::view::cycle` usable in SYCL by removing the pointer field.
- Adding some `constexpr` specifiers.

The limitations of the SYCL programming model also impose certain restrictions on device-code:

- Cannot throw in device-code.
- Cannot use views with pointer fields or non-standard-layout types, without modification.
- Views need to support random-access for parallel usage in SYCL. This means that views such as `filter` and `remove_if` cannot be used as input to SYCL parallel algorithms without first instantiating their result, as they are at most bidirectional.

4 Performance Results

We have run a number of performance measurements to evaluate our implementation. We used an Intel i7-6700K CPU and its integrated Intel HD Graphics 530 GPU.

We use the zero-copy functionality provided by the Intel hardware and OpenCL runtime to eliminate data transfer costs. This requires allocating memory at the 4096 byte boundary and creating the SYCL buffer with the `cl::sycl::using_host_ptr` property.

```
1  const auto height = 512;
2  const auto width = 512;
3  const auto iterations = 100;
4
5  std::vector<pixel> image(height * width);
6
7  {
8   gstorm::sycl_exec exec;
9
10  auto gpu_image =
11    std::experimental::copy(exec, image);
12
13  auto indices = ranges::view::iota(0)
14               | ranges::view::take(
15                   width * height);
16  std::experimental::transform(
17    exec, indices, gpu_image,
18    CalculatePixel{height, width, iterations});
19 }
```

Listing 7. Calculating the mandelbrot set using our prototype.

We run all experiments 100 times and report the median execution time measured on the host (e.g. including any buffer creation or kernel launch overheads).

4.1 Speedup from Fusion

Figure 2 shows speedups from fusing kernel calls using our prototype for 5 benchmarks. The baseline implementation uses our equivalents to the `std::transform` and `std::reduce` standard algorithms and the fast versions have some of them replaced by views. The results show that fusion does not degrade performance and results in speedups of up to 3.5×.

4.2 Calculating the Mandelbrot Set

In particular, we would like to highlight the results we obtained from calculating the mandelbrot set and comparing our implementation to the Intel Parallel STL. [5] Implemented in Listing 7, it takes image pixel locations as input and outputs an image containing a colorised visualisation of the mandelbrot set.

As seen in Figure 3, the SYCL ranges version of mandelbrot is significantly faster than the Intel Parallel STL although the Intel Parallel STL is based on Intel Threading Building Blocks and is highly optimised for CPUs.

This is because the STL does not have a specialised fused function for `std::iota` with `std::transform`. In line 13 of Listing 7 the ranges equivalent of `std::iota` is used to create the pixel indices which are then passed to the function calculating the output. Additionally, sinc there is no parallel version of `std::iota` there have to be two library calls, one of which is always sequential.

5 Conclusion

In this paper, we have presented some problems with the existing parallel algorithms interface when targeting heterogeneous systems, and the current SYCL Parallel STL solutions. We introduce a prototype implementation where we use *range-v3* to show how range-based algorithms, views, and actions leverage many of the problems with the current parallel algorithms' interface, and enable further optimisations, such as kernel fusion, that are not possible with the iterator interface.

We encourage the C++ Standardisation community to consider including views and actions in the next C++ Standard to facilitate adoption of C++ on heterogeneous platforms. Ranges without views and actions are insufficient to address the problems faced on these platforms.

We would like to ask the authors of the Ranges TS to consider making some requirements for standard-layout types whenever possible. Standard-layout types are currently the only guarantee that a programming model for heterogeneous systems can enforce so that data can be shared across different compilers and ABIs. Non-standard-layout types cannot be copied to a device 'as-is' for which code is compiled with a different toolchain.

Finally, as an intermediate step, we encourage the Library Evolution Working Group to consider adding a mechanism for identifying contiguous iterators to the Standard for C++, so that Parallel STL implementations can detect whether iterator ranges are contiguous and can assume that data can be directly and continuously accessed through a pointer. We also encourage LEWG to consider extending the Ranges TS to support a `ContiguousIterator` and a `ContiguousRange` concept.

6 Future Work

Future work will continue to explore the combination of parallel algorithms with ranges, with special attention paid to fusion. We would like to explore other topics, such as data layout transformation, and concept definitions that would be meaningful for parallel algorithm implementations that target non-CPU architectures.

The authors, and the SYCL group in Khronos, will continue to work with the SYCL Parallel STL implementation, exploring the different issues that heterogeneous computing present to the C++ standard. We are looking forward to feedback to our ideas but also more general collaborations on any aspect that may facilitate the programmability of heterogeneous systems in C++.

Finally, this work has identified that typical implementations of `std::tuple` may not be suitable for heterogeneous programming. We believe there is a problem with allowing `std::tuple` to be non-standard-layout, which has been sufficiently exposed by our endeavour to marry parallel programming and the Ranges TS, and we would like to investigate ways to refine `std::tuple` – and possibly other types – so that they become suitable for heterogeneous programming.

References

[1] 2017. Prototype of SYCL Parallel STL with Ranges. https://github.com/codeplaysoftware/parallelts-range-exploration.

[2] Christopher Di Bella. 2018. Christopher Di Bella's answer to 'What do you think of the C++'s Range TS?'. http://bit.ly/2nsUvvu

[3] Gordon Brown, Christopher Di Bella, Michael Haidl, Toomas Remmelg, Ruyman Reyes, Michel Steuwer, and Michael Wong. 2018. P0836R0: Introduce Parallelism to the Ranges TS. http://wg21.link/p0836r0

[4] Casey Carter. 2015. cmcstl2. https://github.com/CaseyCarter/cmcstl2

[5] Intel Corporation. 2017. Parallel STL. https://github.com/intel/parallelstl

[6] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[7] Michael Haidl, Michel Steuwer, Hendrik Dirks, Tim Humernbrum, and Sergei Gorlatch. 2017. Towards Composable GPU Programming: Programming GPUs with Eager Actions and Lazy Views. In *PMAM*. ACM.

[8] ISO/IEC JTC1/SC22/WG21. 2017. ISO/IEC 21425:2017, Programming Languages – C++ Extensions for Ranges.

[9] Eric Niebler. 2014. range-v3. https://github.com/ericniebler/range-v3

[10] Eric Niebler. 2017. P0789R1: Range Adaptors and Utilities. http://wg21.link/p0789r1

[11] Eric Niebler, Sean Parent, and Andrew Sutton. 2014. N4128: Ranges for the Standard Library, Revision 1. http://wg21.link/n4128

[12] Ruyman Reyes, Gordon Brown, Michael Wong, and Hartmut Kaiser. 2017. P0687R0: Data Movement in C++. http://wg21.link/p0687r0